FIG. 1

```
                100
struct S {
    S() throw();   101
    ~S() throw();   102
};
                103
struct T {
   T();   104
    ~T();   105
};

void woof();
             106
...
L1: {
    T ant;   107
108  try {       109
        if( x>0 ) {
            S boa;   110
 111    } else {
            S cat;   112
            T dog;   113
            woof();   114
 115    }
116  } catch( int y ) {   117
        S elk;   118
        woof();   119
120  }
  }   121
L2:;
```

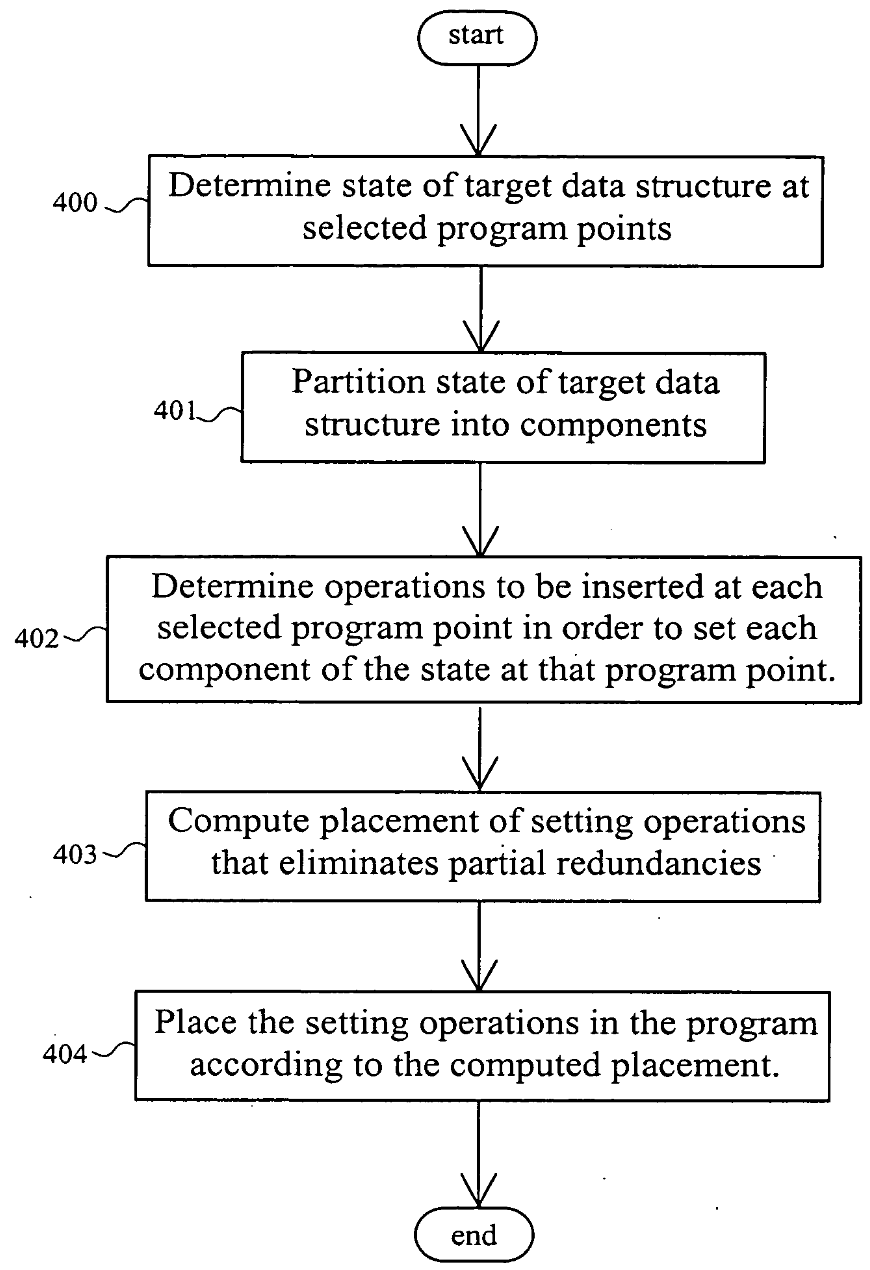
```
#include <setjmp.h>
                  200
struct EH_item {              201
    struct EH_item * next;
    enum {DESTROY,TRY} tag;  202
    union {
        struct {
            void * object;  203
            void (*dtor)();  204
        } destructor;
        struct {
            jmp_buf buffer;  205
            struct handler_spec* handlers;  206
        } try_block;
    };
};

struct EH_item * EH_stack_ptr;  207
```

FIG. 2

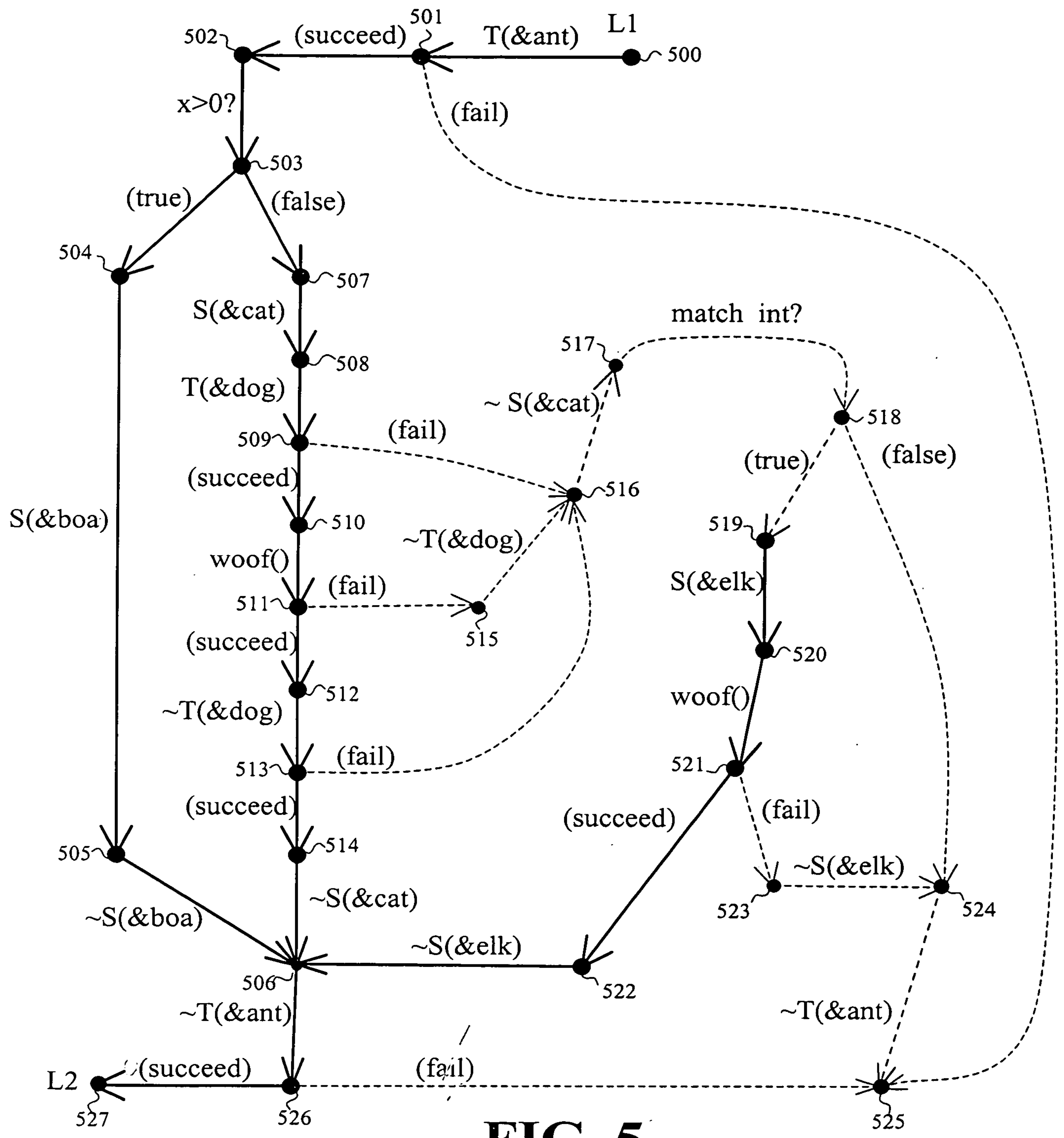
FIG. 3
PRIOR ART

```
         struct EH_item ra, rb, rc, rd, re, rt;
         L1:
303      T(&ant);
304      ra.kind = DESTROY;
         ra.destructor.object  = &ant; ra.destructor.dtor = &~T;
306      ra.next = EH_stack_ptr;  EH_stack_ptr = &ra;
307      rt.kind = TRY;
         rt.next = EH_stack_ptr;
         rt.try_block.handlers = ...;
31       rt.next = EH_stack_ptr;  EH_stack_ptr = &rt;
31       if( setjmp( rt.try_block.buffer)==0 ) {
             if( x>0 ) {
         313     S(&boa);
         314     rb.kind = DESTROY;
                 rb.destructor.object  = &boa; rb.destructor.dtor = &~S;
                 rb.next = EH_stack_ptr;  EH_stack_ptr = &rb;
         31      EH_stack_ptr = EH_stack_ptr->next;
                 ~S(& boa);
             } else {
                 S(cat);
                 rc.kind = DESTROY;
                 rc.destructor.object  = &cat; rc.destructor.dtor = &~S;
                 rc.next = EH_stack_ptr;   EH_stack_ptr = &rc;
                 T(&dog);
                 rd.kind = DESTROY;
                 rd.destructor.object  = &dog; rd.destructor.dtor = & ~T;
                 rd.next = EH_stack_ptr;   EH_stack_ptr = &rd;
                 woof();
                 EH_stack_ptr = EH_stack_ptr->next;
                 ~T(&dog);
                 EH_stack_ptr = EH_stack_ptr->next;
                 ~S(& cat);
             }
         } else {
             S(&elk);
             re.kind = DESTROY;
             re.destructor.object  = &elk; re.destructor.dtor = address of ~S();
             re.next = EH_stack_ptr;  EH_stack_ptr = &re;
             ~S(&elk);
             EH_stack_ptr = EH_stack_ptr->next;
         }
342      EH_stack_ptr = EH_stack_ptr->next;
343      EH_stack_ptr = EH_stack_ptr->next;
34       ~T(ant);
         L2:
```



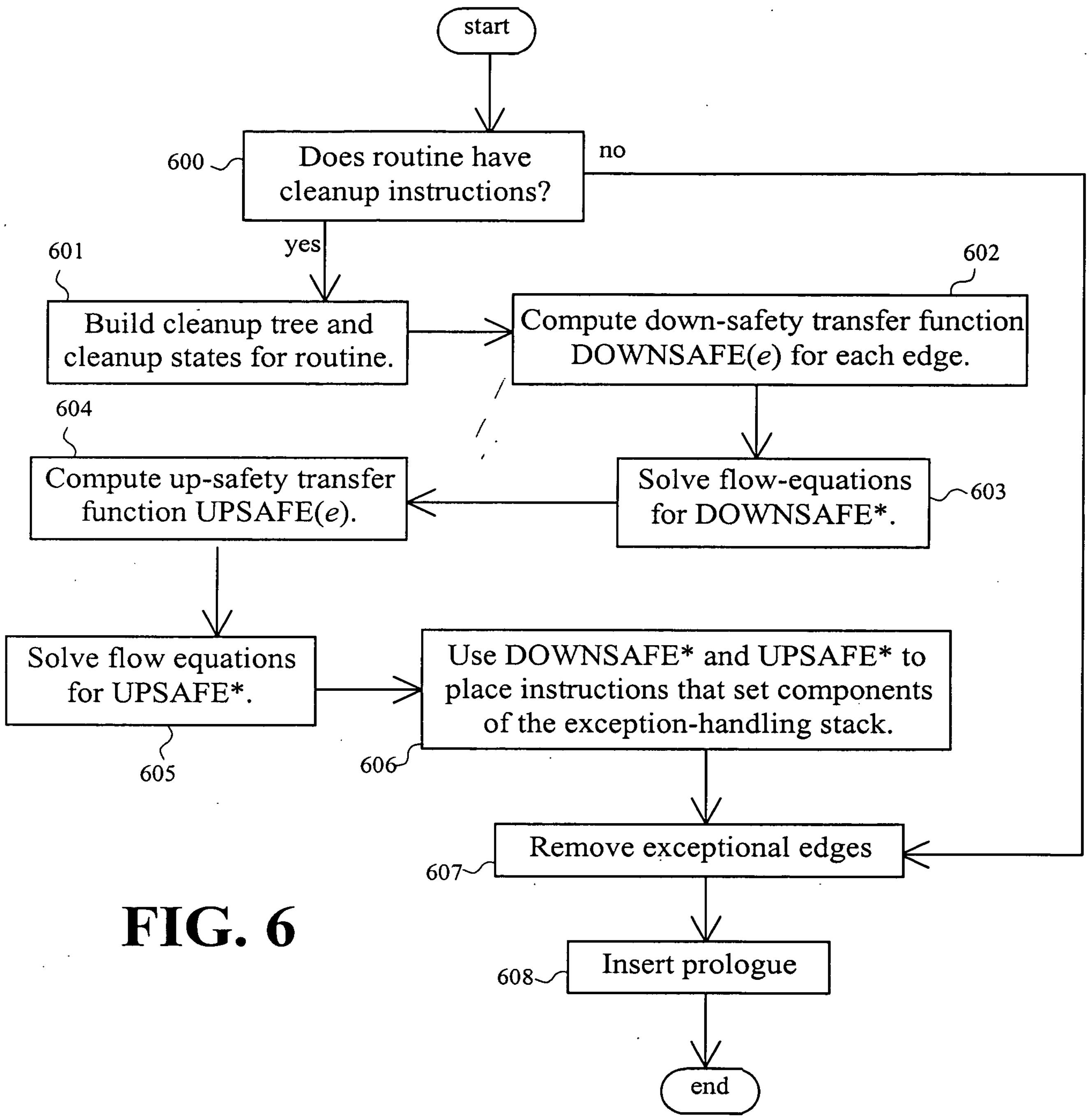
start
400
Determine state of target data structure at selected program points
401
Partition state of target data structure into components
402
Determine operations to be inserted at each selected program point in order to set each component of the state at that program point.
403
Compute placement of setting operations that eliminates partial redundancies
404
Place the setting operations in the program according to the computed placement.
end

FIG. 4



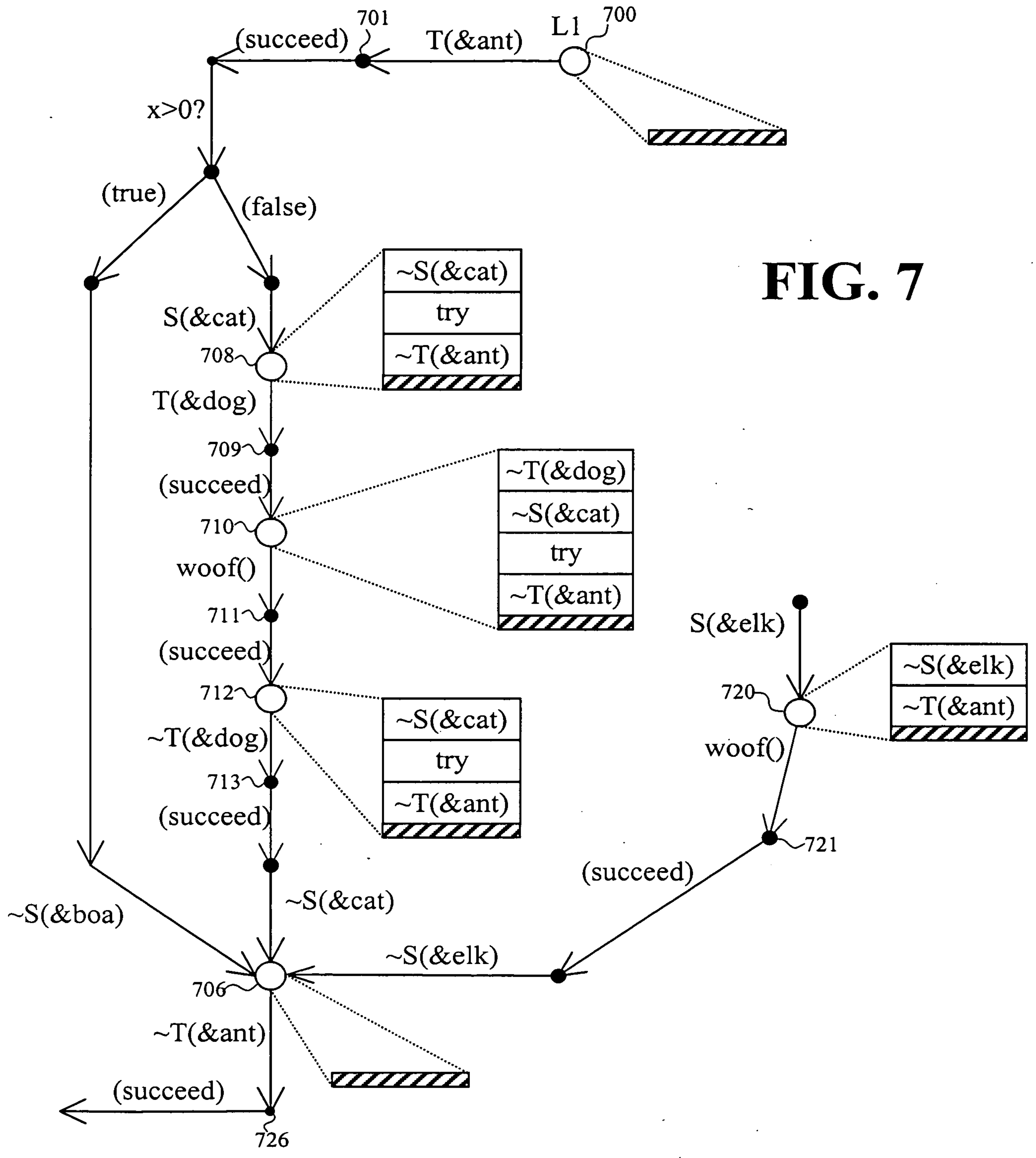
L1
500
T(&ant)
501
(succeed)
502
(fail)
x>0?
503
(true)
(false)
504
507
S(&cat)
508
T(&dog)
509
(fail)
(succeed)
510
woof()
511
(fail)
515
~T(&dog)
516
~ S(&cat)
517
match int?
518
(true)
(false)
519
S(&elk)
520
woof()
521
(succeed)
(fail)
523
~S(&elk)
524
(succeed)
512
~T(&dog)
513
(fail)
(succeed)
514
S(&boa)
505
~S(&boa)
~S(&cat)
~S(&elk)
522
506
~T(&ant)
~T(&ant)
L2
(succeed)
(fail)
527
526
525

FIG. 5



start
600
Does routine have cleanup instructions?
no
yes
601
Build cleanup tree and cleanup states for routine.
602
Compute down-safety transfer function DOWNSAFE(e) for each edge.
604
Compute up-safety transfer function UPSAFE(e).
Solve flow-equations for DOWNSAFE*.
603
Solve flow equations for UPSAFE*.
605
Use DOWNSAFE* and UPSAFE* to place instructions that set components of the exception-handling stack.
606
Remove exceptional edges
607
Insert prologue
608
end

FIG. 6



L1
700
T(&ant)
701
(succeed)
x>0?
(true)
(false)
S(&cat)
708
~S(&cat)
try
~T(&ant)
T(&dog)
709
(succeed)
710
~T(&dog)
~S(&cat)
try
~T(&ant)
woof()
711
(succeed)
712
~S(&cat)
try
~T(&ant)
~T(&dog)
713
(succeed)
~S(&cat)
~S(&boa)
S(&elk)
720
~S(&elk)
~T(&ant)
woof()
721
(succeed)
~S(&elk)
706
~T(&ant)
(succeed)
726

FIG. 7

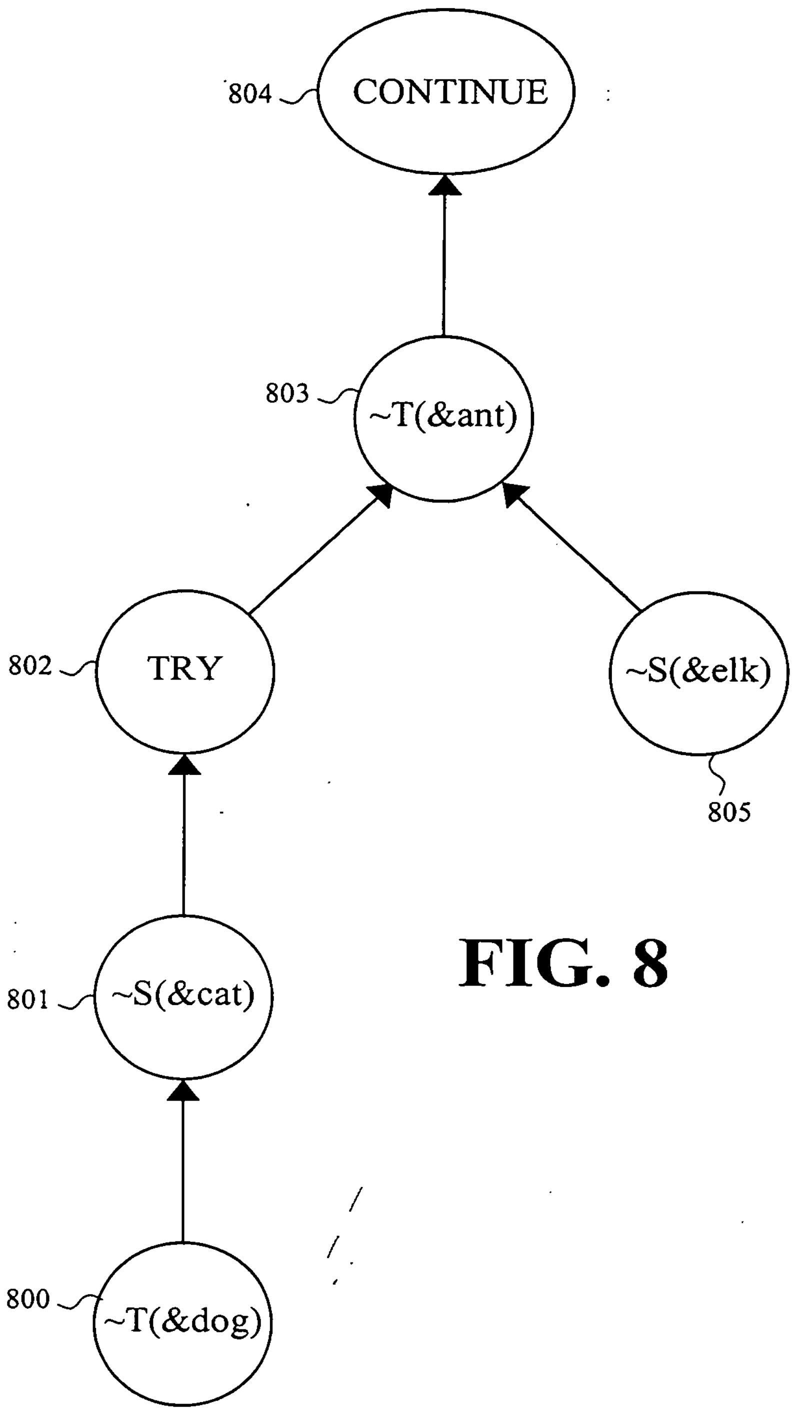
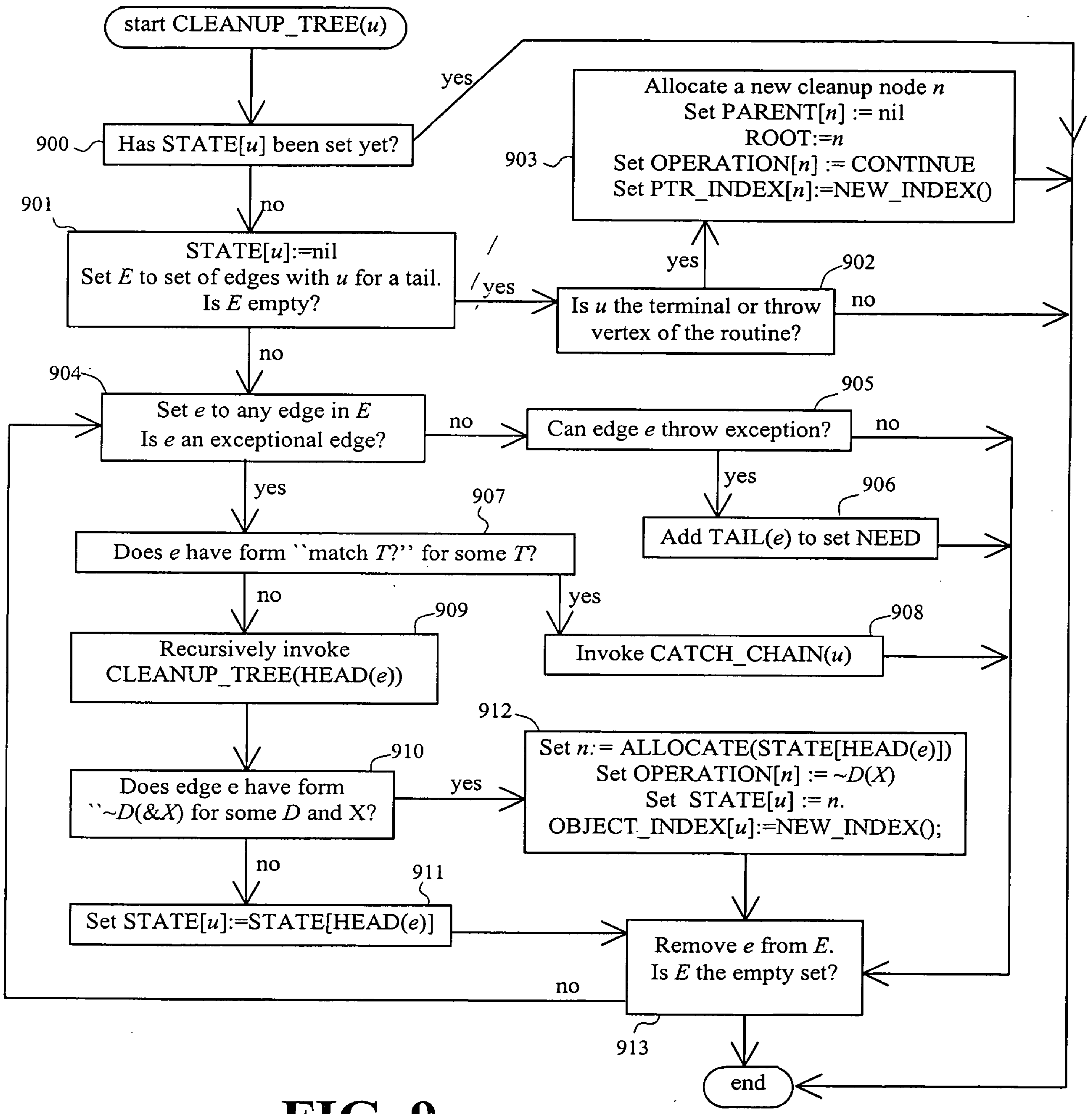
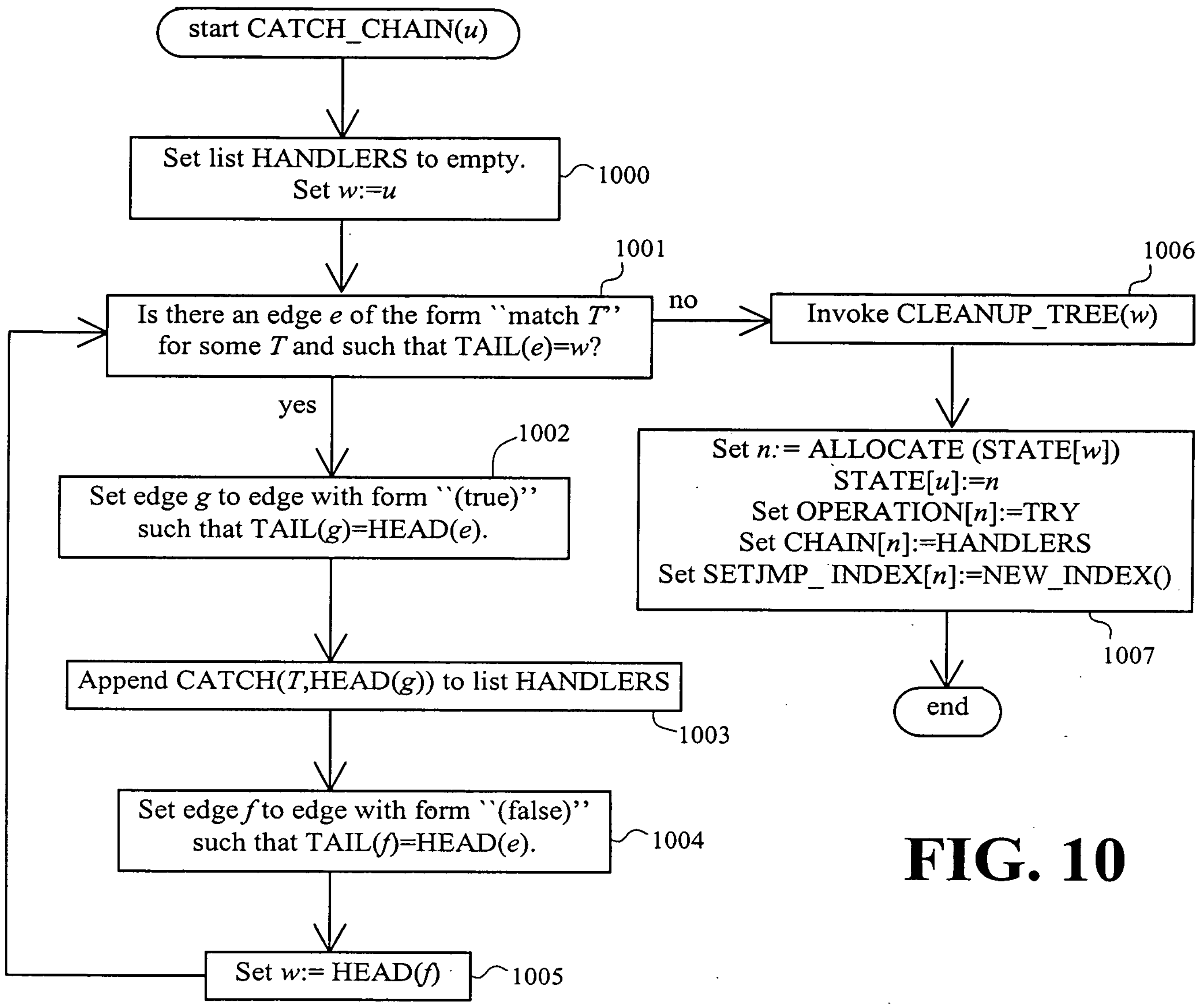


FIG. 8



start CLEANUP_TREE(u)
900
Has STATE[u] been set yet?
yes
no
901
STATE[u]:=nil
Set E to set of edges with u for a tail.
Is E empty?
yes
902
Is u the terminal or throw vertex of the routine?
yes
no
903
Allocate a new cleanup node n
Set PARENT[n] := nil
ROOT:=n
Set OPERATION[n] := CONTINUE
Set PTR_INDEX[n]:=NEW_INDEX()
no
904
Set e to any edge in E
Is e an exceptional edge?
no
905
Can edge e throw exception?
no
yes
906
Add TAIL(e) to set NEED
yes
907
Does e have form ``match T?'' for some T?
no
yes
909
Recursively invoke CLEANUP_TREE(HEAD(e))
908
Invoke CATCH_CHAIN(u)
910
Does edge e have form ``~D(&X) for some D and X?
yes
912
Set n:= ALLOCATE(STATE[HEAD(e)])
Set OPERATION[n] := ~D(X)
Set STATE[u] := n.
OBJECT_INDEX[u]:=NEW_INDEX();
no
911
Set STATE[u]:=STATE[HEAD(e)]
Remove e from E.
Is E the empty set?
no
913
end

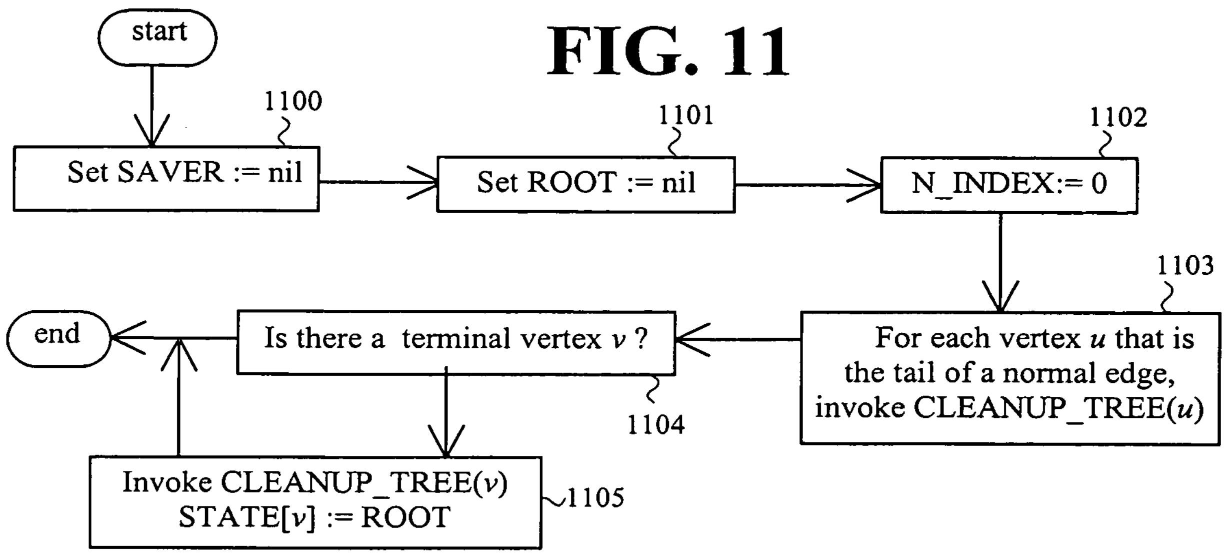
FIG. 9



start CATCH_CHAIN(u)
Set list HANDLERS to empty. Set w:=u
1000
Is there an edge e of the form ``match T'' for some T and such that TAIL(e)=w?
1001
no
yes
Invoke CLEANUP_TREE(w)
1006
Set edge g to edge with form ``(true)'' such that TAIL(g)=HEAD(e).
1002
Set n:= ALLOCATE (STATE[w])
STATE[u]:=n
Set OPERATION[n]:=TRY
Set CHAIN[n]:=HANDLERS
Set SETJMP_ INDEX[n]:=NEW_INDEX()
1007
Append CATCH(T,HEAD(g)) to list HANDLERS
1003
end
Set edge f to edge with form ``(false)'' such that TAIL(f)=HEAD(e).
1004
Set w:= HEAD(f)
1005

FIG. 10

FIG. 11



start
1100
Set SAVER := nil
1101
Set ROOT := nil
1102
N_INDEX:= 0
1103
For each vertex u that is the tail of a normal edge, invoke CLEANUP_TREE(u)
Is there a terminal vertex v ?
1104
end
Invoke CLEANUP_TREE(v) STATE[v] := ROOT
1105

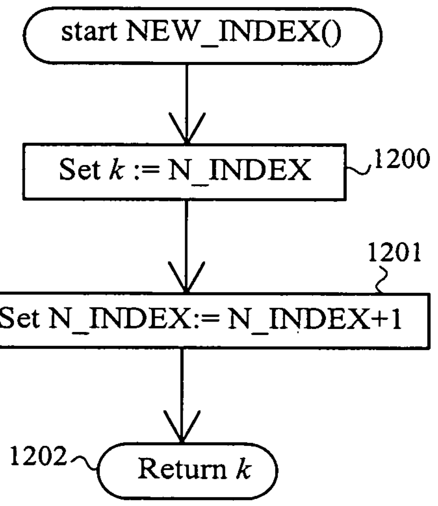


FIG. 12

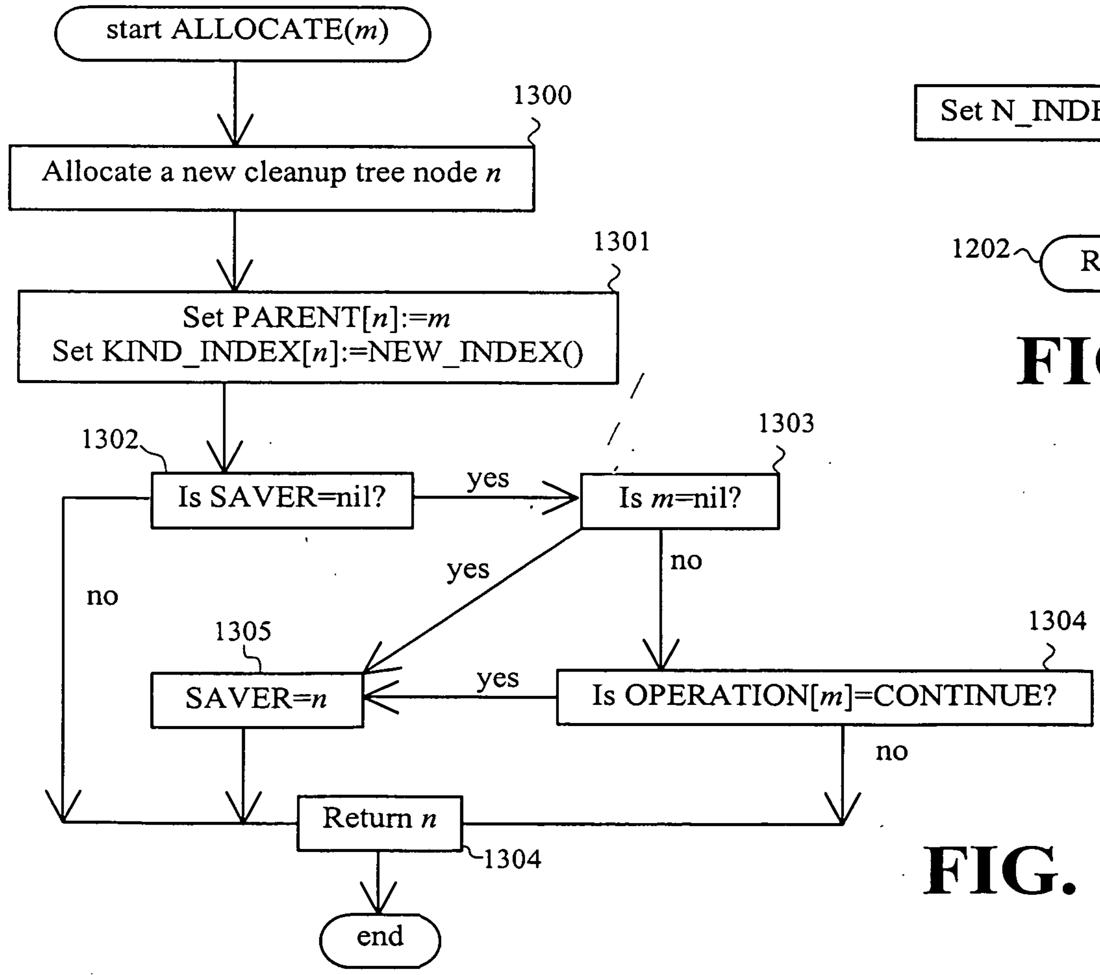


FIG. 13

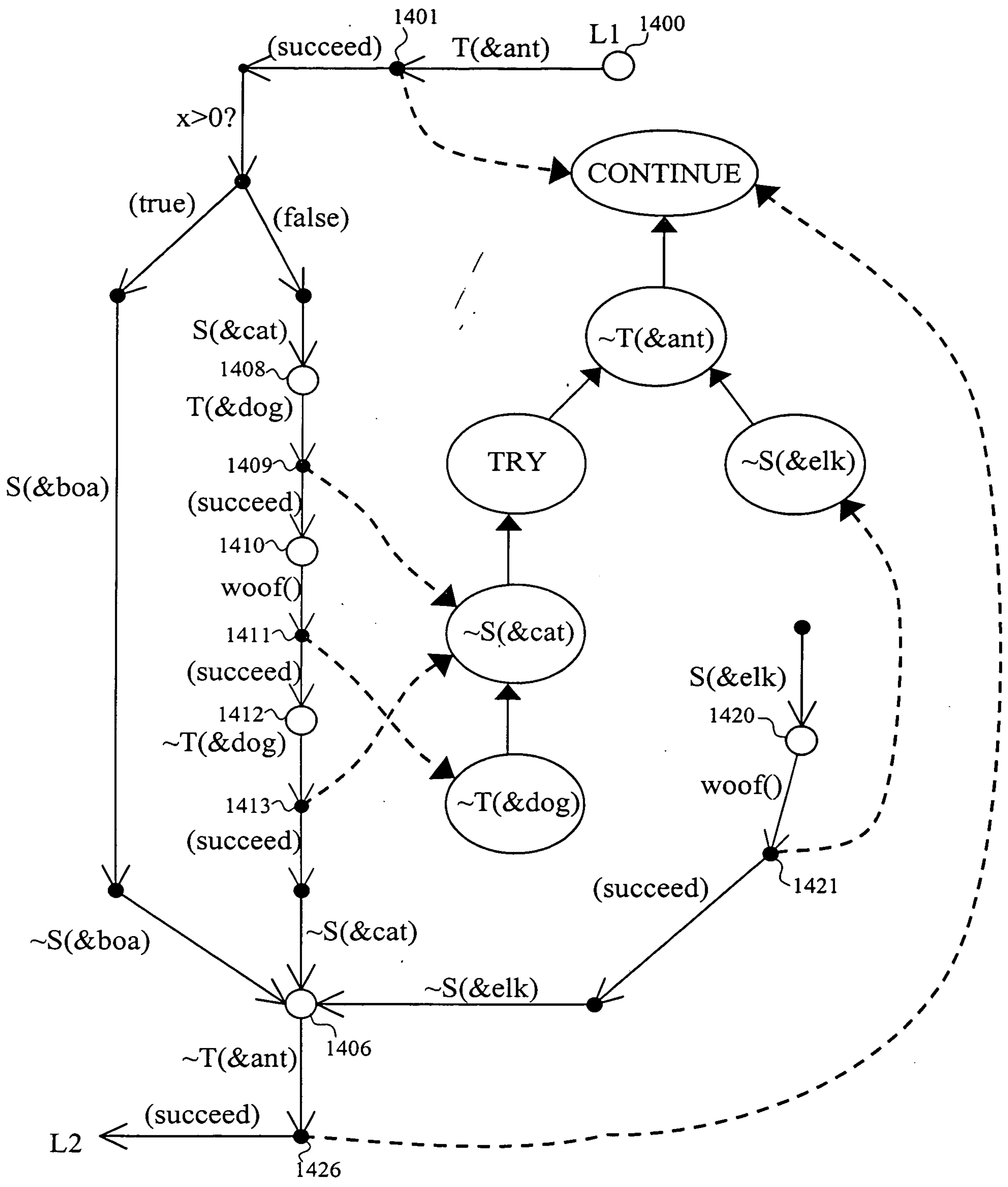
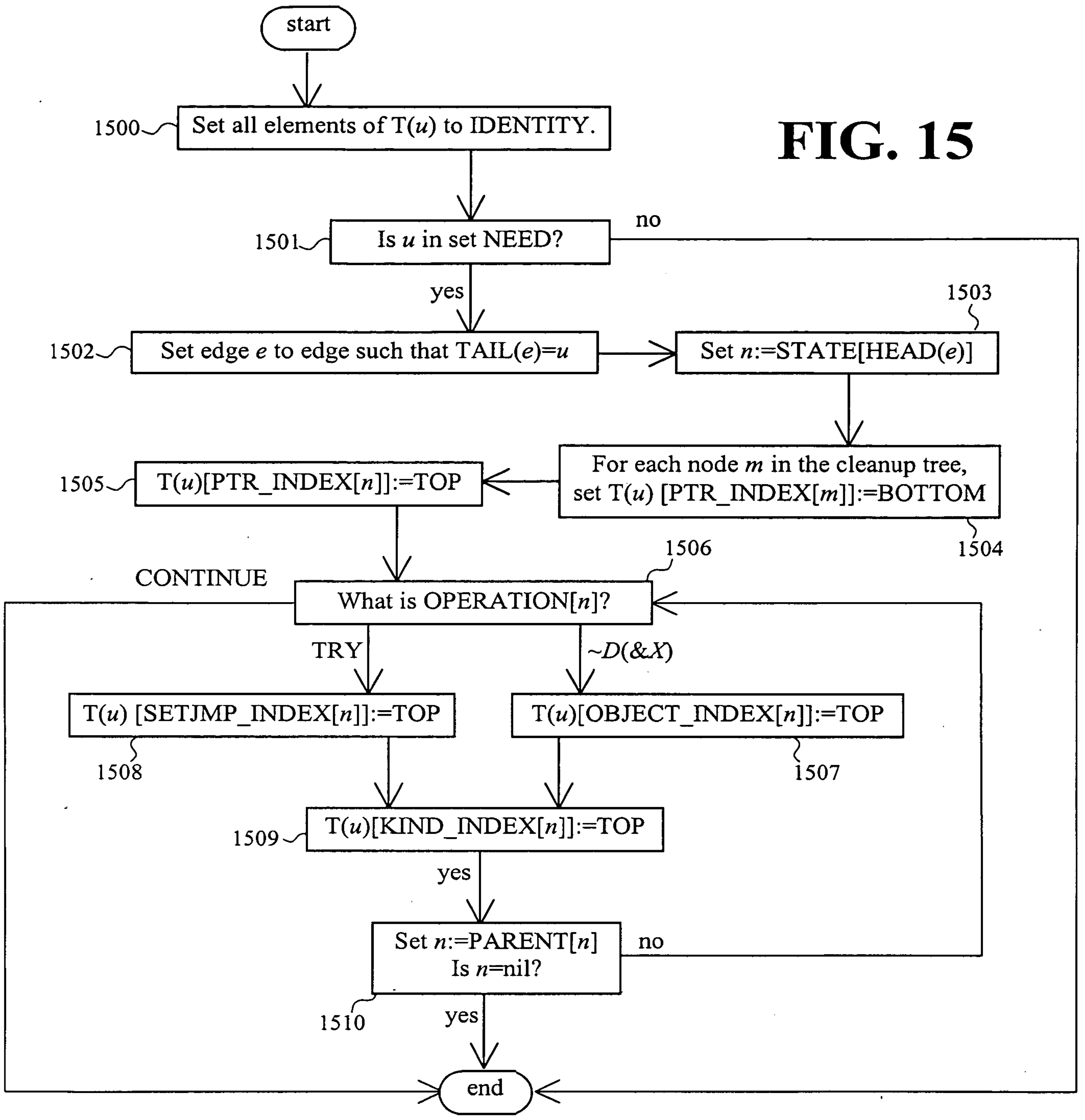

1401
(succeed)
T(&ant)
L1
1400
x>0?
CONTINUE
(true)
(false)
S(&cat)
~T(&ant)
1408
T(&dog)
TRY
~S(&elk)
1409
S(&boa)
(succeed)
1410
woof()
~S(&cat)
1411
(succeed)
S(&elk)
1412
1420
~T(&dog)
woof()
1413
~T(&dog)
(succeed)
1421
(succeed)
~S(&boa)
~S(&cat)
~S(&elk)
1406
~T(&ant)
(succeed)
L2
1426

FIG. 14



start
1500
Set all elements of T(u) to IDENTITY.
1501
Is u in set NEED?
no
yes
1502
Set edge e to edge such that TAIL(e)=u
1503
Set n:=STATE[HEAD(e)]
For each node m in the cleanup tree, set T(u) [PTR_INDEX[m]]:=BOTTOM
1504
1505
T(u)[PTR_INDEX[n]]:=TOP
CONTINUE
1506
What is OPERATION[n]?
TRY
~D(&X)
T(u) [SETJMP_INDEX[n]]:=TOP
1508
T(u)[OBJECT_INDEX[n]]:=TOP
1507
1509
T(u)[KIND_INDEX[n]]:=TOP
yes
Set n:=PARENT[n] Is n=nil?
no
1510
yes
end

FIG. 15

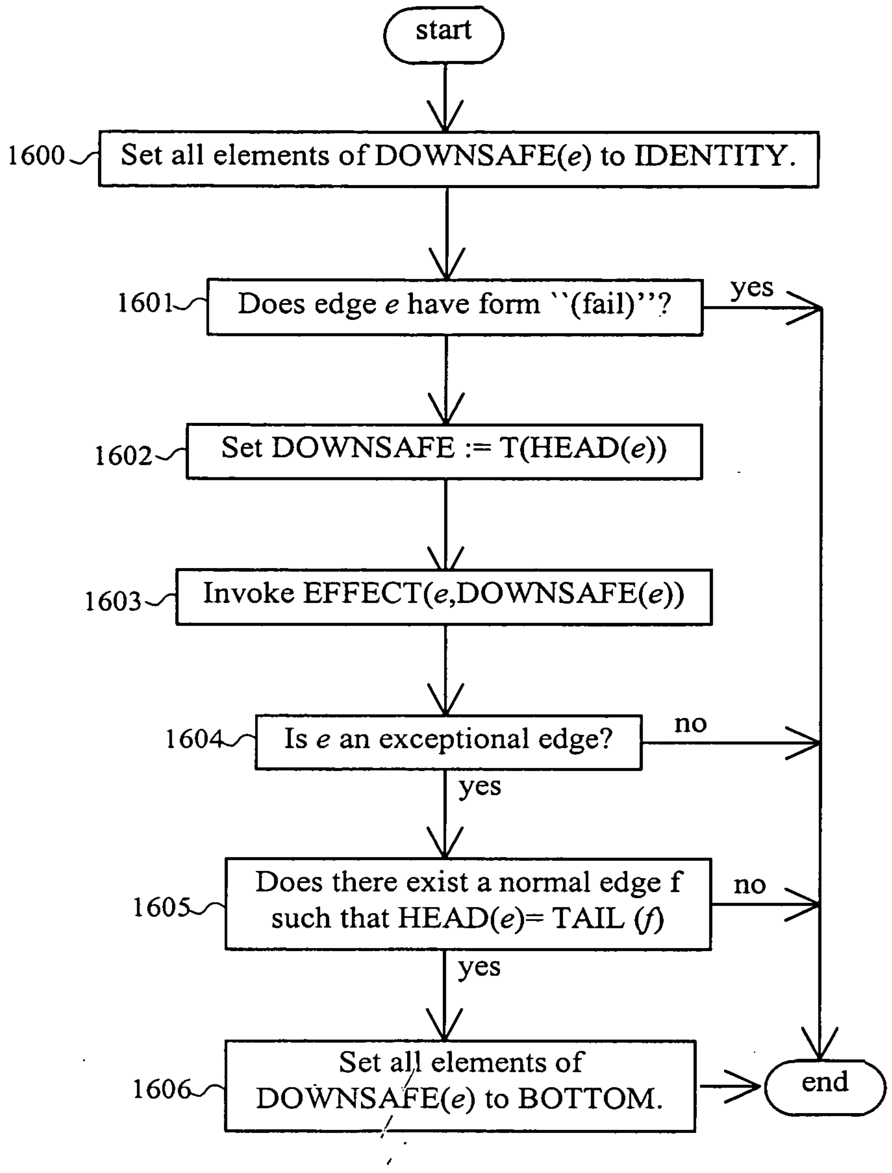

start
1600
Set all elements of DOWNSAFE(e) to IDENTITY.
1601
Does edge e have form ``(fail)''?
yes
1602
Set DOWNSAFE := T(HEAD(e))
1603
Invoke EFFECT(e,DOWNSAFE(e))
1604
Is e an exceptional edge?
no
yes
1605
Does there exist a normal edge f such that HEAD(e)= TAIL (f)
no
yes
1606
Set all elements of DOWNSAFE(e) to BOTTOM.
end

FIG. 16

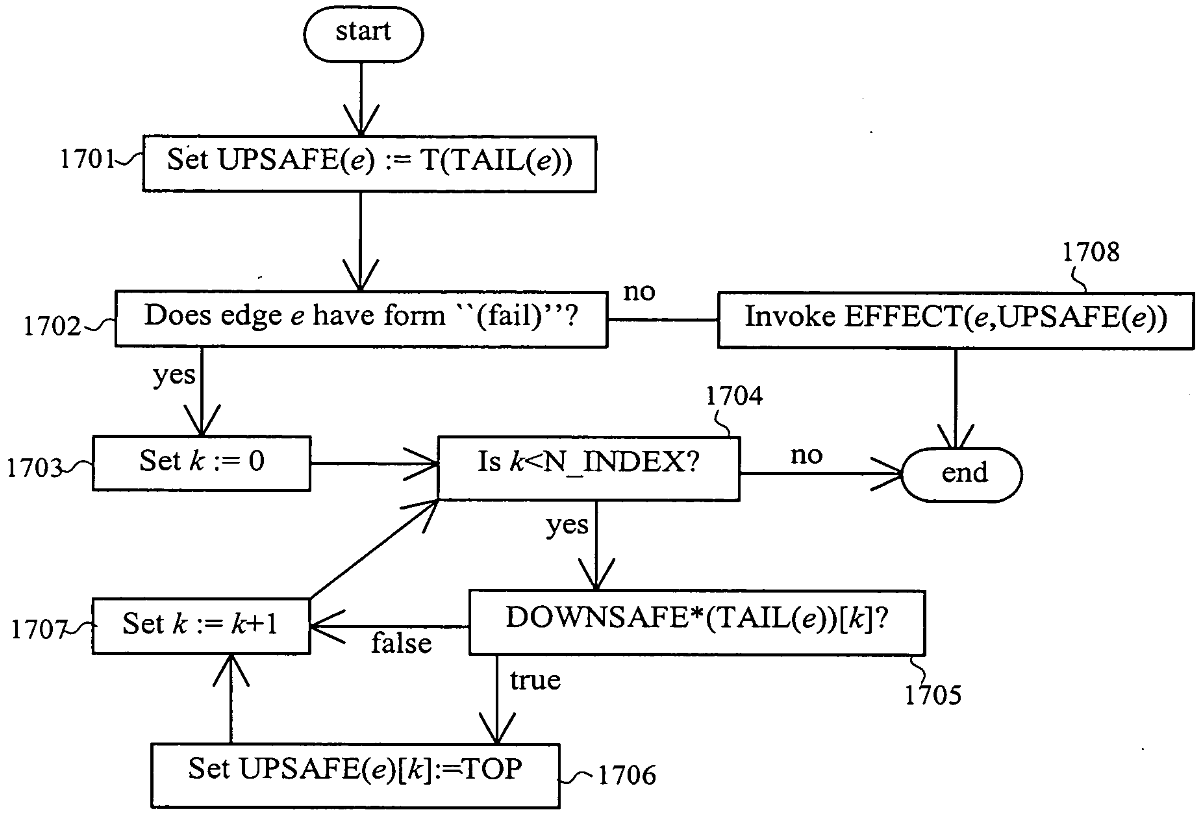
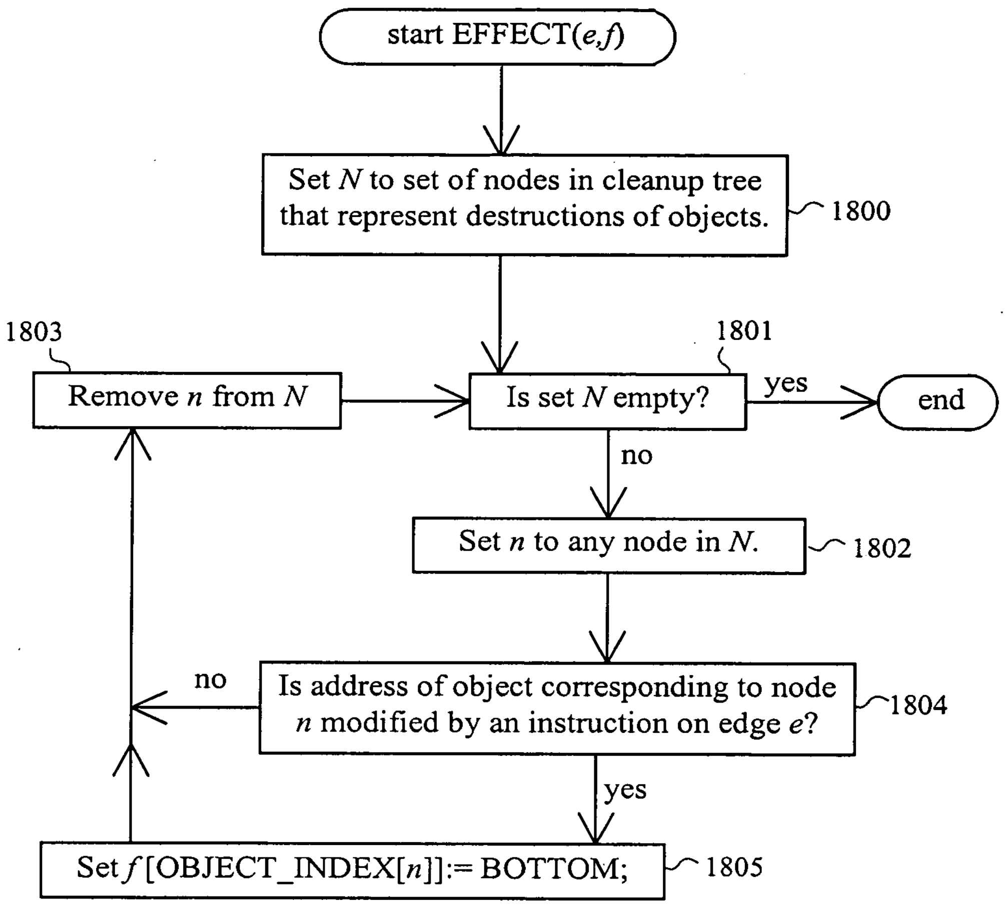
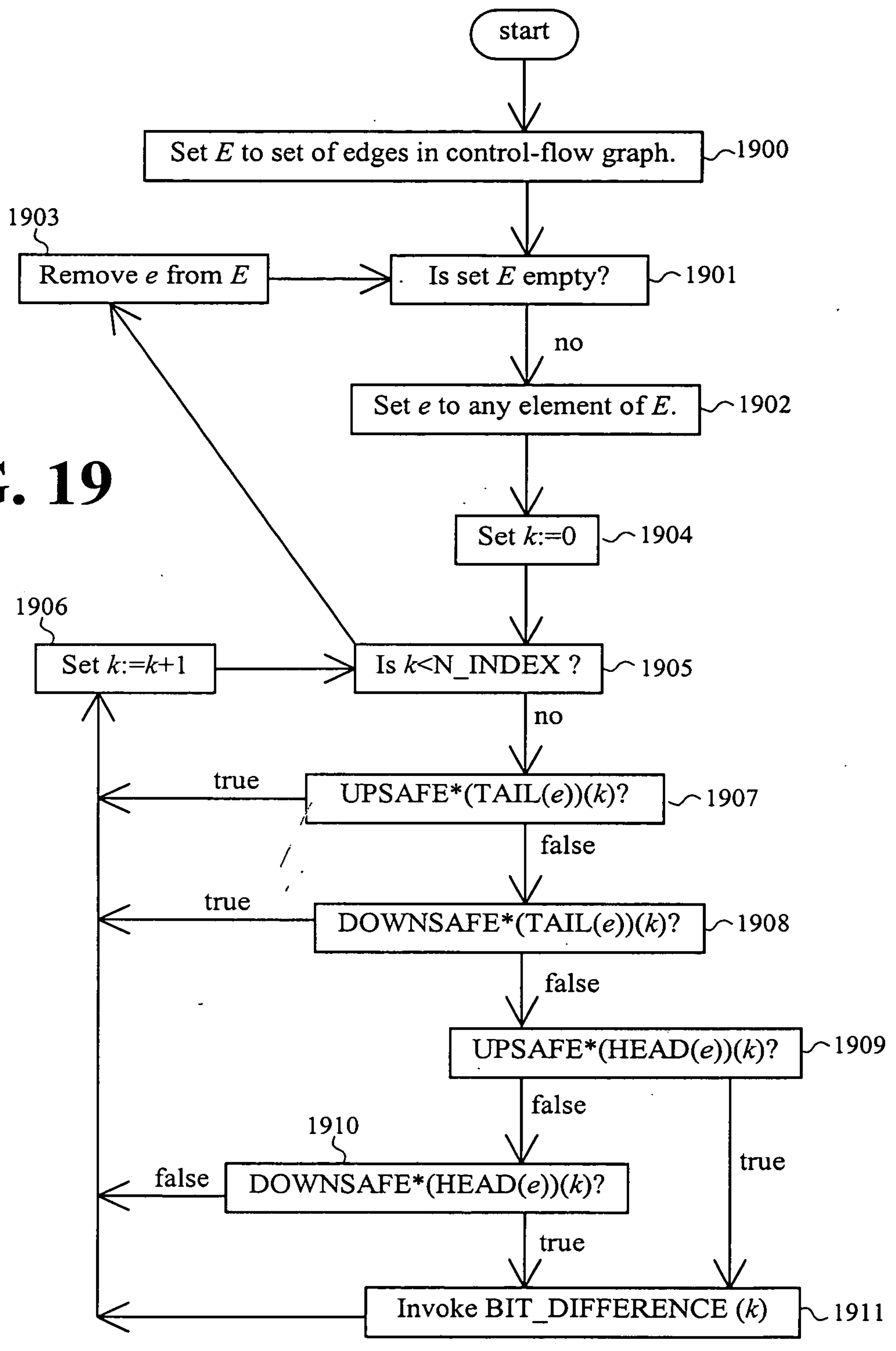

start
1701
Set UPSAFE(e) := T(TAIL(e))
1702
Does edge e have form ``(fail)''?
no
1708
Invoke EFFECT(e,UPSAFE(e))
yes
1703
Set k := 0
1704
Is k<N_INDEX?
no
end
yes
1707
Set k := k+1
false
DOWNSAFE*(TAIL(e))[k]?
1705
true
Set UPSAFE(e)[k]:=TOP
1706

FIG. 17



start EFFECT(*e*,*f*)
Set *N* to set of nodes in cleanup tree that represent destructions of objects.
1800
1803
1801
Remove *n* from *N*
Is set *N* empty?
yes
end
no
Set *n* to any node in *N*.
1802
no
Is address of object corresponding to node *n* modified by an instruction on edge *e*?
1804
yes
Set *f* [OBJECT_INDEX[*n*]]:= BOTTOM;
1805

FIG. 18



start
Set E to set of edges in control-flow graph.
1900
Remove e from E
1903
Is set E empty?
1901
no
Set e to any element of E.
1902
Set k:=0
1904
Set k:=k+1
1906
Is k<N_INDEX ?
1905
no
UPSAFE*(TAIL(e))(k)?
1907
true
false
DOWNSAFE*(TAIL(e))(k)?
1908
true
false
UPSAFE*(HEAD(e))(k)?
1909
false
true
DOWNSAFE*(HEAD(e))(k)?
1910
false
true
Invoke BIT_DIFFERENCE (k)
1911

FIG. 19

FIG. 20

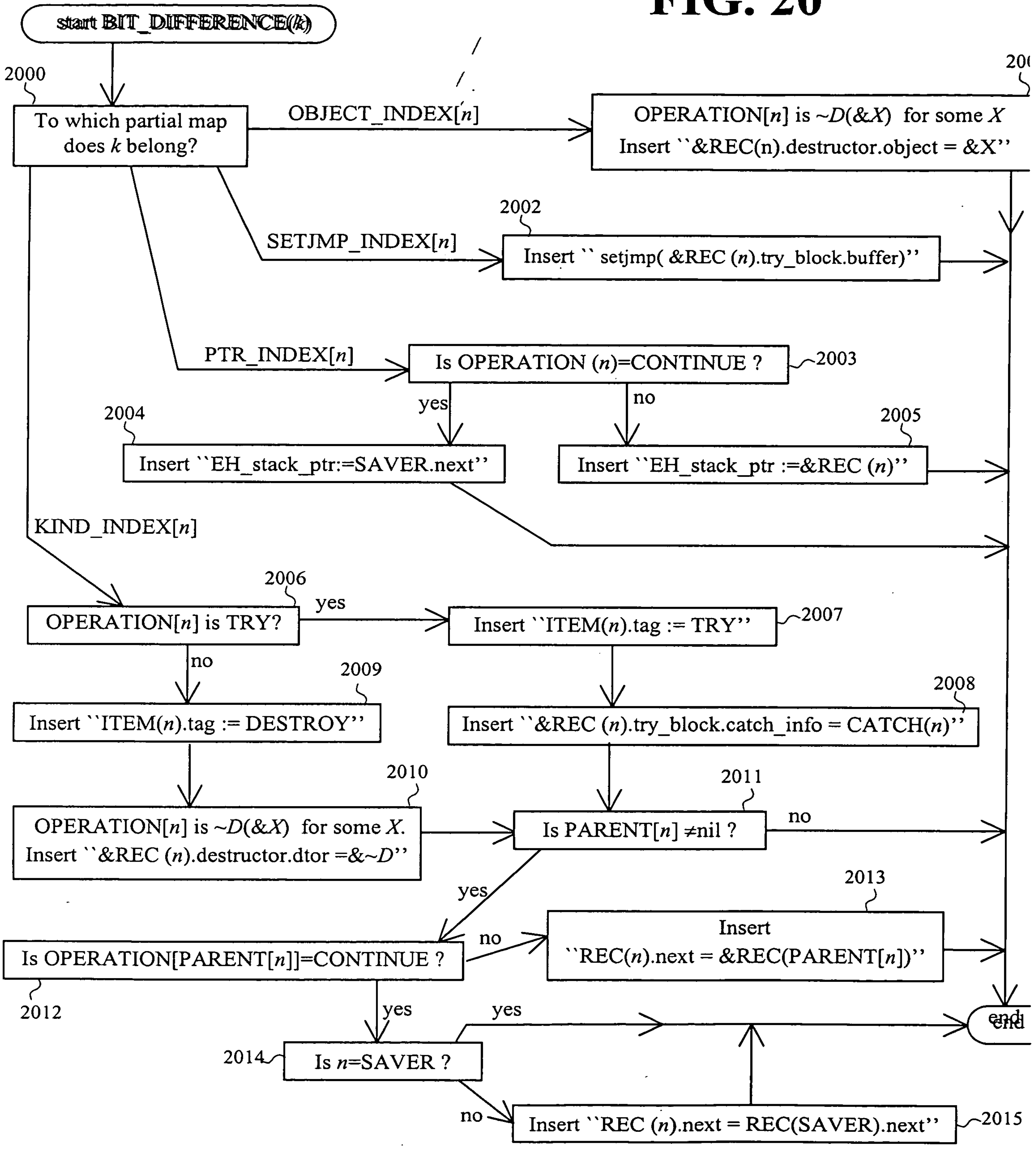

start BIT_DIFFERENCE(k)
2000
To which partial map does k belong?
OBJECT_INDEX[n]
OPERATION[n] is ~D(&X) for some X
Insert ``&REC(n).destructor.object = &X''
2002
SETJMP_INDEX[n]
Insert `` setjmp(&REC (n).try_block.buffer)''
PTR_INDEX[n]
Is OPERATION (n)=CONTINUE ?
2003
yes
no
2004
Insert ``EH_stack_ptr:=SAVER.next''
2005
Insert ``EH_stack_ptr :=&REC (n)''
KIND_INDEX[n]
2006
OPERATION[n] is TRY?
yes
Insert ``ITEM(n).tag := TRY''
2007
no
2009
Insert ``ITEM(n).tag := DESTROY''
2008
Insert ``&REC (n).try_block.catch_info = CATCH(n)''
2010
OPERATION[n] is ~D(&X) for some X.
Insert ``&REC (n).destructor.dtor =&~D''
2011
Is PARENT[n] ≠nil ?
no
yes
2013
Insert
``REC(n).next = &REC(PARENT[n])''
no
Is OPERATION[PARENT[n]]=CONTINUE ?
2012
yes
yes
end
2014
Is n=SAVER ?
no
Insert ``REC (n).next = REC(SAVER).next''
2015

FIG. 21

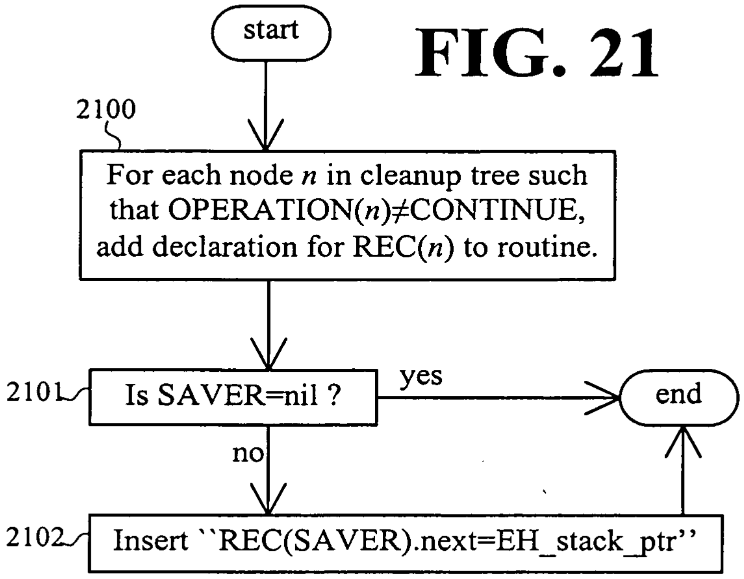
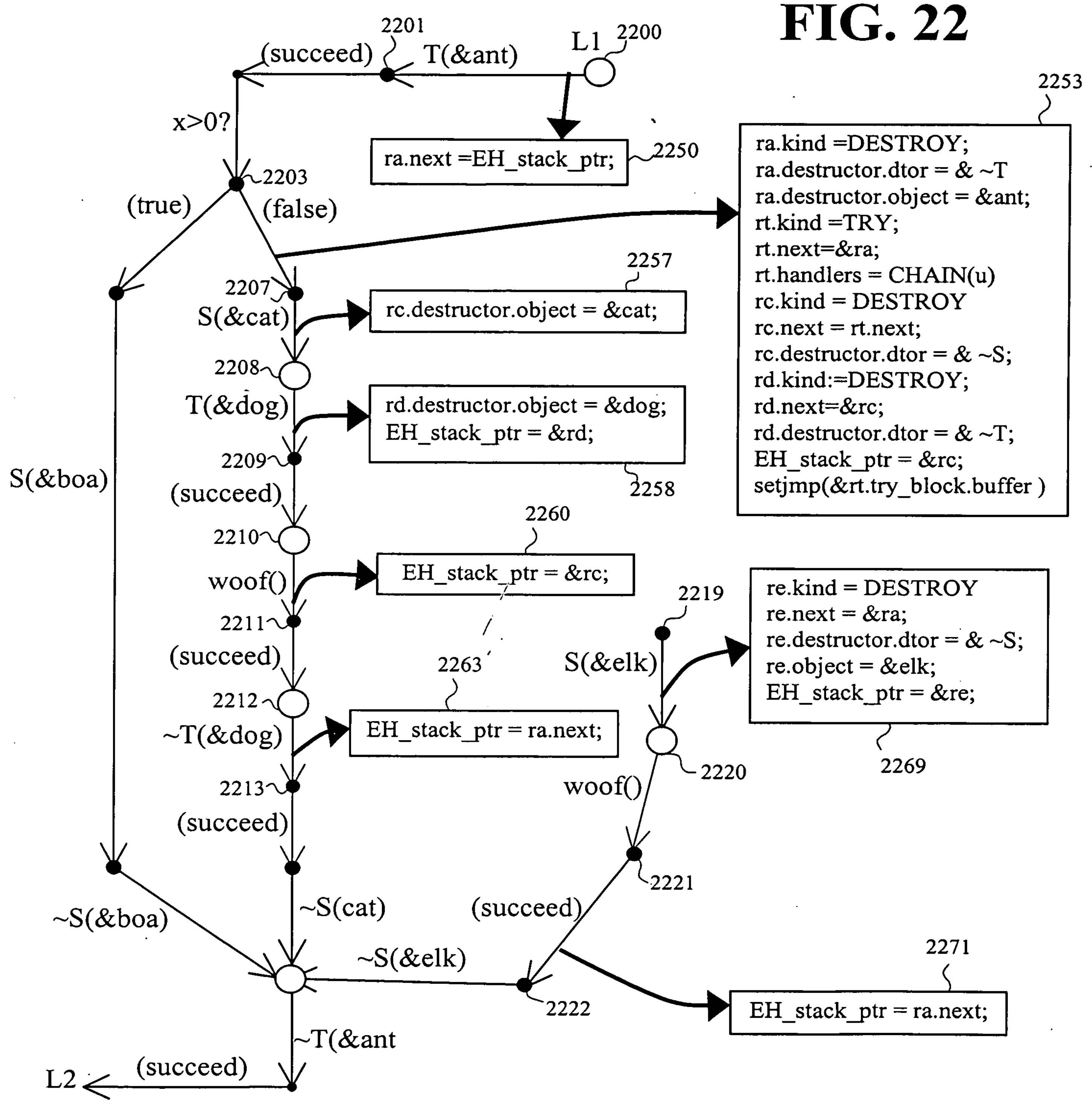

start
2100
For each node *n* in cleanup tree such that OPERATION(*n*)≠CONTINUE, add declaration for REC(*n*) to routine.
2101
Is SAVER=nil ?
yes
end
no
2102
Insert ``REC(SAVER).next=EH_stack_ptr''

FIG. 22



L1
2200
2201
T(&ant)
(succeed)
x>0?
2203
(true)
(false)
ra.next =EH_stack_ptr;
2250
2253
ra.kind =DESTROY;
ra.destructor.dtor = & ~T
ra.destructor.object = &ant;
rt.kind =TRY;
rt.next=&ra;
rt.handlers = CHAIN(u)
rc.kind = DESTROY
rc.next = rt.next;
rc.destructor.dtor = & ~S;
rd.kind:=DESTROY;
rd.next=&rc;
rd.destructor.dtor = & ~T;
EH_stack_ptr = &rc;
setjmp(&rt.try_block.buffer)
2207
S(&cat)
2257
rc.destructor.object = &cat;
2208
T(&dog)
rd.destructor.object = &dog;
EH_stack_ptr = &rd;
2258
2209
(succeed)
2210
2260
woof()
EH_stack_ptr = &rc;
2211
(succeed)
2263
2212
~T(&dog)
EH_stack_ptr = ra.next;
2213
(succeed)
S(&boa)
~S(&boa)
~S(cat)
2219
S(&elk)
re.kind = DESTROY
re.next = &ra;
re.destructor.dtor = & ~S;
re.object = &elk;
EH_stack_ptr = &re;
2269
2220
woof()
2221
(succeed)
~S(&elk)
2222
2271
EH_stack_ptr = ra.next;
~T(&ant
(succeed)
L2

FIG. 23

```
struct EH_item ra, rb, rc, rd, re, rt;~2301
L1:
ra.next = EH_stack_ptr;~2303
T(&ant);
if( x>0 ) {
      S(&boa);
      ~S(&boa);
} else {
     ra.kind = DESTROY;
     ra.destructor.dtor = &~T;
     ra.destructor.object  = &ant;
     rt.kind = TRY;
     rt.next = &ra;
     rt.try_block.handlers = ...;
     rt.next = &ra;
     rc.kind = DESTROY;
     rc.destructor.dtor = &~S;
     rc.next = &rt;
     rd.kind = DESTROY;
     rd.destructor.dtor = &~T;
     rd.next = &rc;
     eh_stack_ptr = &rc;
     if( setjmp( &rt.try_block.buffer)==0 ) {
          S(&cat);
          rc.destructor.object  = &cat;
          T(&dog)
          rd.destructor.object  = &dog;
          eh_stack_ptr = &rd;
          woof();
          EH_stack_ptr = &rc;
          ~T(&dog);
          EH_stack_ptr = ra.next;
           ~S(&cat);
     } else {
     re.kind = DESTROY;
          re.next = &ra;
          re.destructor.dtor = &~S;
          eh_stack_ptr = &re;
          S(&elk);
          re.destructor.object  = &elk;
          woof();
          EH_stack_ptr = ra.next
          ~S(&elk);
     }
}
~T&ant)
L2:
```

FIG. 24

```
struct R {
   R(); ~2402
   ~R() throw(); ~2403
};
...
{
    i=0;
    do { ~2408
       R fox; ~2409
       woof(); ~2410
       i=i+1;
    } while( i<100 ); ~2412
}
```

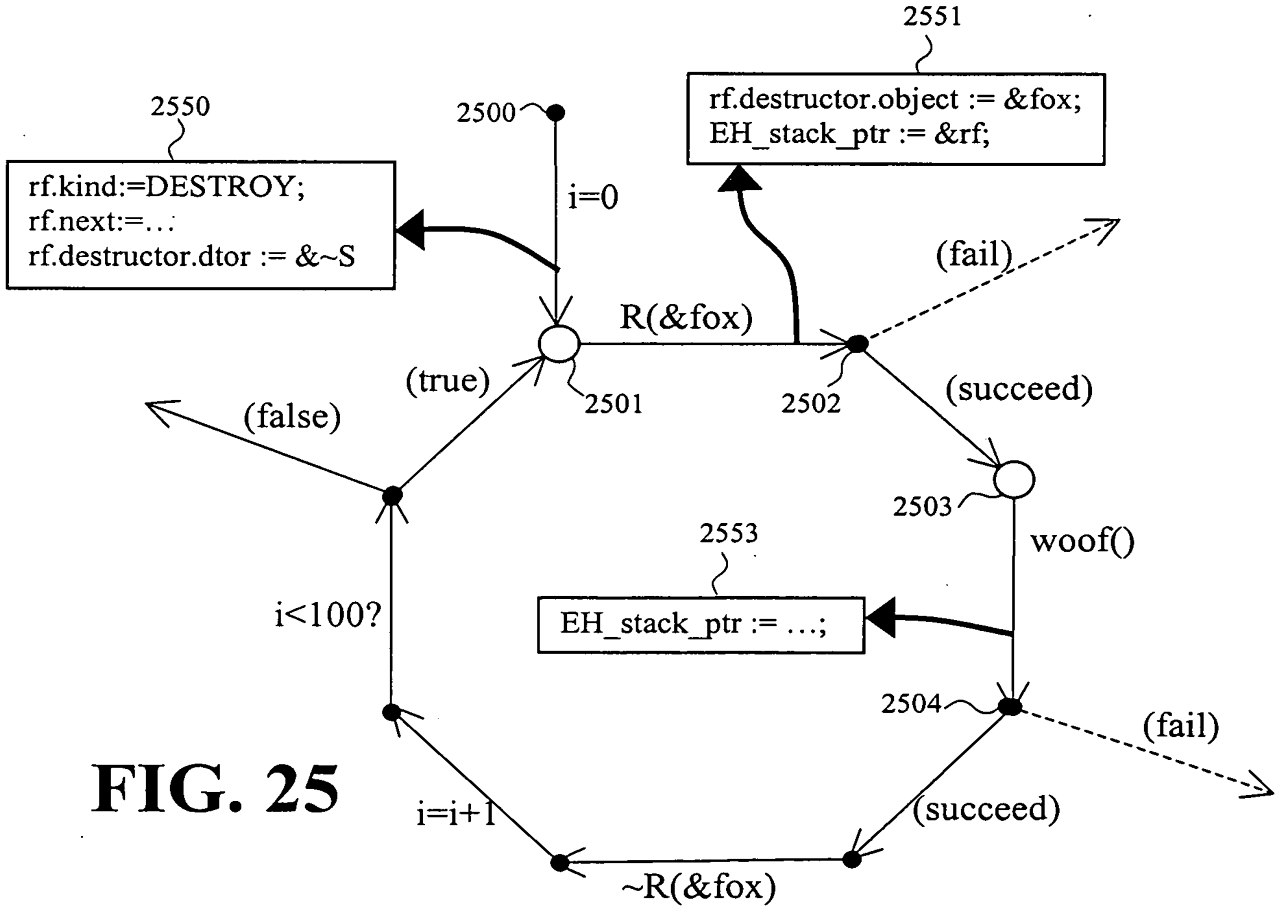
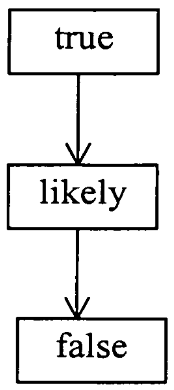


FIG. 25

FIG. 26


true
likely
false

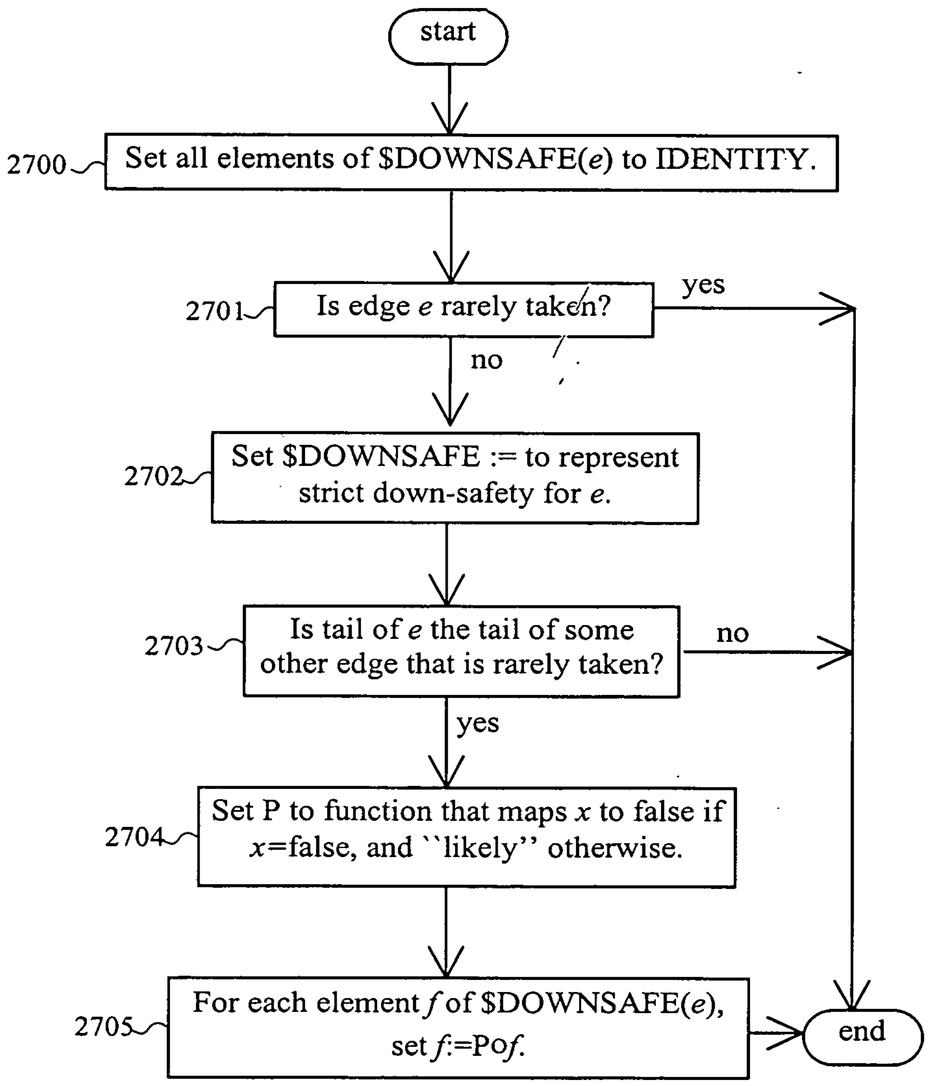
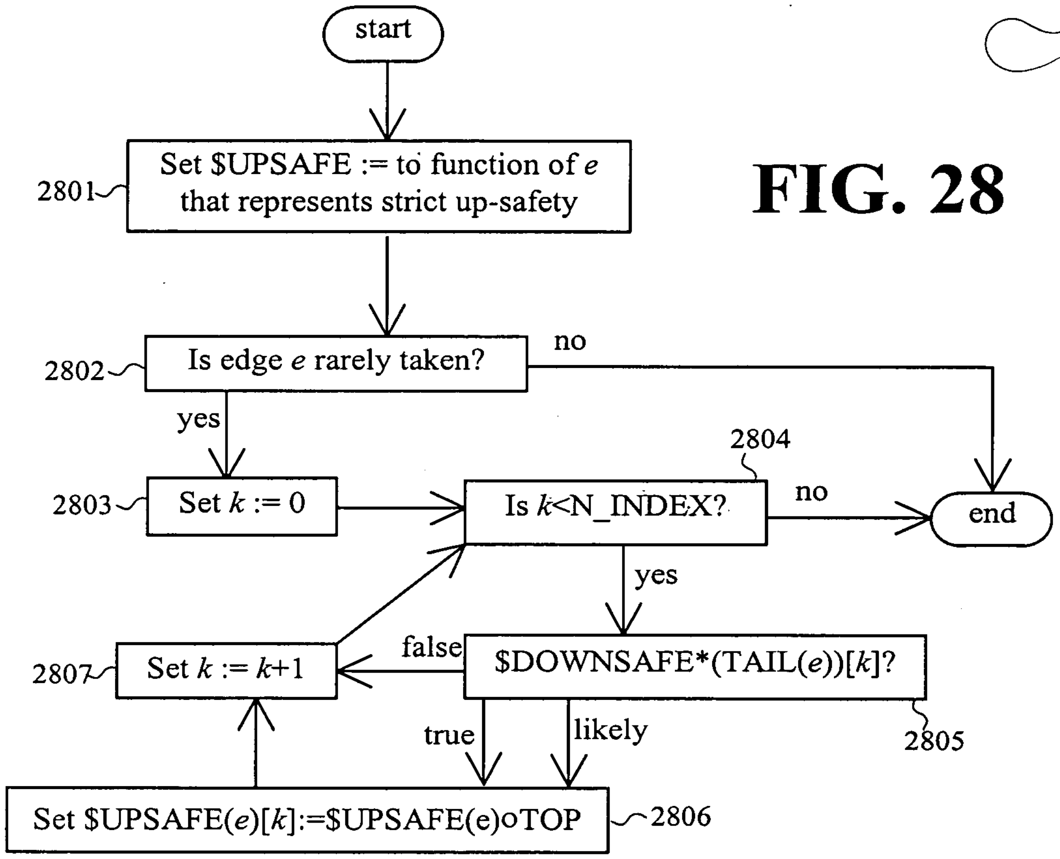

start
2700
Set all elements of $DOWNSAFE(e) to IDENTITY.
2701
Is edge e rarely taken?
yes
no
2702
Set $DOWNSAFE := to represent strict down-safety for e.
2703
Is tail of e the tail of some other edge that is rarely taken?
no
yes
2704
Set P to function that maps x to false if x=false, and ``likely'' otherwise.
2705
For each element f of $DOWNSAFE(e), set f:=Pof.
end

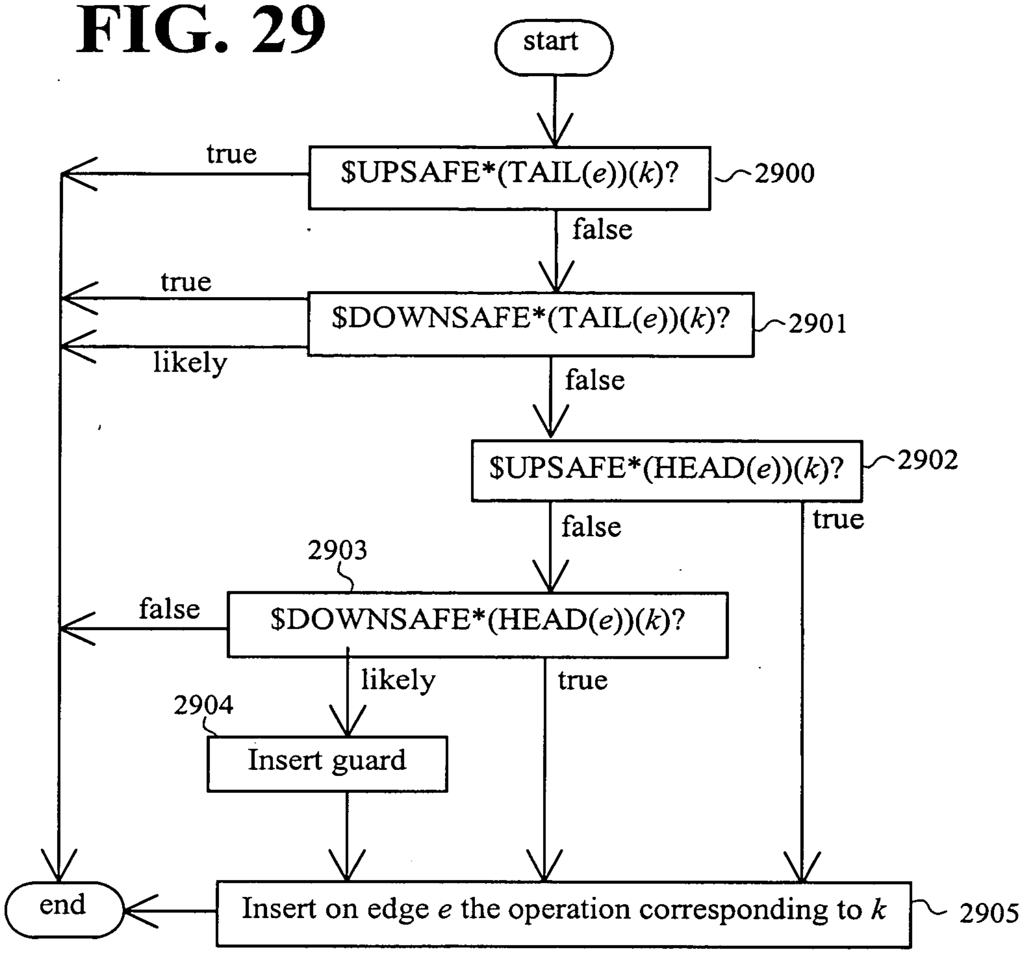
FIG. 27



start
2801
Set $UPSAFE := to function of e that represents strict up-safety
2802
Is edge e rarely taken?
no
yes
2803
Set k := 0
2804
Is k<N_INDEX?
no
end
yes
2807
Set k := k+1
false
$DOWNSAFE*(TAIL(e))[k]?
2805
true
likely
Set $UPSAFE(e)[k]:=$UPSAFE(e)oTOP
2806

FIG. 28

FIG. 29


start
$UPSAFE*(TAIL(e))(k)?
2900
true
false
$DOWNSAFE*(TAIL(e))(k)?
2901
true
likely
false
$UPSAFE*(HEAD(e))(k)?
2902
false
true
2903
$DOWNSAFE*(HEAD(e))(k)?
false
likely
true
2904
Insert guard
end
Insert on edge e the operation corresponding to k
2905

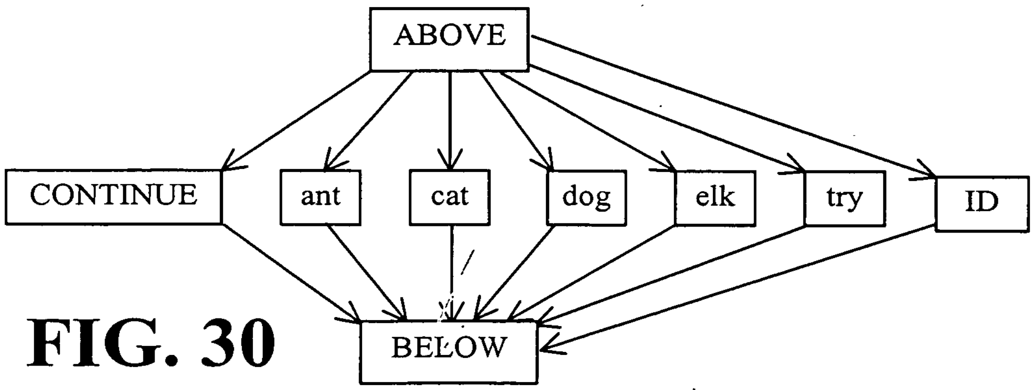

ABOVE
CONTINUE
ant
cat
dog
elk
try
ID
BELOW

FIG. 30